DELISLE Laure
3A - Classe 33

Projet programmation C

---LABYRINTHE---

Enseignant : Michael FRANCOIS

Année 2014/2015 – Semestre 1

Introduction

Sujet :

L'objectif de ce projet est de modéliser en C, un labyrinthe en 2D. Ensuite, faire une recherche d'un chemin quelconque entre l'entrée et la sortie, puis le plus court chemin dans ce labyrinthe. On considère le labyrinthe comme un tableau bidimensionnel d'entiers courts, de taille donnée. On doit pouvoir générer le labyrinthe de plusieurs façons :

1. En fixant dès le départ tous les paramètres du labyrinthe (taille du tableau, entiers, position de l'entrée, etc.), afin de bien débuter et bien contrôler votre programme.
2. Générer aléatoirement un labyrinthe avec toutes ses caractéristiques de base.
3. On doit pouvoir également charger un labyrinthe depuis un fichier texte, contenant toutes les caractéristiques nécessaires. Le format de fichier choisi sera le suivant :
    - La 1ère ligne contiendra le nombre de lignes et de colonnes du labyrinthe, puis les coordonnées de l'entrée et de la sortie du labyrinthe.
    - Ensuite, le fichier contiendra un tableau de valeurs indiquant la configuration initiale de chaque cellule.

Organisation personnelle :

J'ai organisé mon travail en 5 phases :

1. Recherches préliminaires
2. Ecriture des specs et de la structure fonctionnelle
3. Implémentation
4. Résolution de problèmes
5. Effets stylistiques

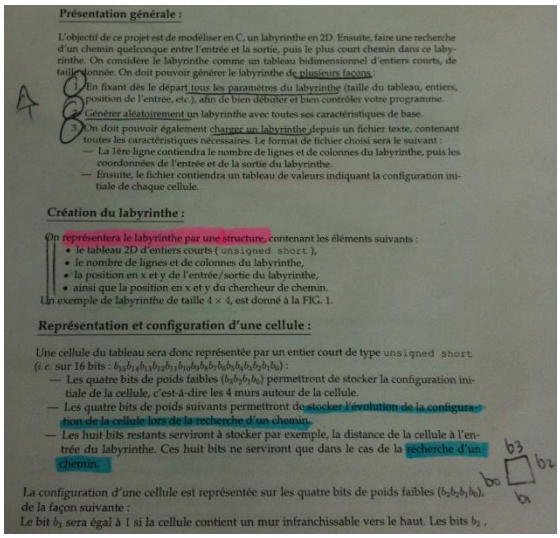
Le fil de mon rapport suivra donc ces phases par ordre chronologique.

Table des matières

I. Recherches préliminaires

Compréhension du sujet

Avant tout, j'ai pris soin de bien lire le sujet. Ainsi, j'ai surligné certaines phrases, j'ai annoté le texte afin de ne pas passer à côté d'une consigne lors de la réalisation.



Présentation générale :

L'objectif de ce projet est de modéliser en C, un labyrinthe en 2D. Ensuite, faire une recherche d'un chemin quelconque entre l'entrée et la sortie, puis le plus court chemin dans ce labyrinthe. On considère le labyrinthe comme un tableau bidimensionnel d'entiers courts, de taille donnée. On doit pouvoir générer le labyrinthe de plusieurs façons :

1. En fixant dès le départ tous les paramètres du labyrinthe (taille du tableau, entiers, position de l'entrée, etc.), afin de bien débuter et bien contrôler votre programme.
2. Générer aléatoirement un labyrinthe avec toutes ses caractéristiques de base.
3. On doit pouvoir également charger un labyrinthe depuis un fichier texte, contenant toutes les caractéristiques nécessaires. Le format de fichier choisi sera le suivant :
   - La 1ère ligne contiendra le nombre de lignes et de colonnes du labyrinthe, puis les coordonnées de l'entrée et de la sortie du labyrinthe.
   - Ensuite, le fichier contiendra un tableau de valeurs indiquant la configuration initiale de chaque cellule.

Création du labyrinthe :

On représentera le labyrinthe par une structure, contenant les éléments suivants :

- le tableau 2D d'entiers courts (`unsigned short`),
- le nombre de lignes et de colonnes du labyrinthe,
- la position en x et y de l'entrée/sortie du labyrinthe,
- ainsi que la position en x et y du chercheur de chemin.

Un exemple de labyrinthe de taille 4 × 4, est donné à la FIG. 1.

Représentation et configuration d'une cellule :

Une cellule du tableau sera donc représentée par un entier court de type `unsigned short` (*i.e.* sur 16 bits : $b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$) :

- Les quatre bits de poids faibles ($b_3b_2b_1b_0$) permettront de stocker la configuration initiale de la cellule, c'est-à-dire les 4 murs autour de la cellule.
- Les quatre bits de poids suivants permettront de stocker l'évolution de la configuration de la cellule lors de la recherche d'un chemin.
- Les huit bits restants serviront à stocker par exemple, la distance de la cellule à l'entrée du labyrinthe. Ces huit bits ne serviront que dans le cas de la recherche d'un chemin.

La configuration d'une cellule est représentée sur les quatre bits de poids faibles ($b_3b_2b_1b_0$), de la façon suivante :
Le bit b_3 sera égal à 1 si la cellule contient un mur infranchissable vers le haut. Les bits b_2 ,

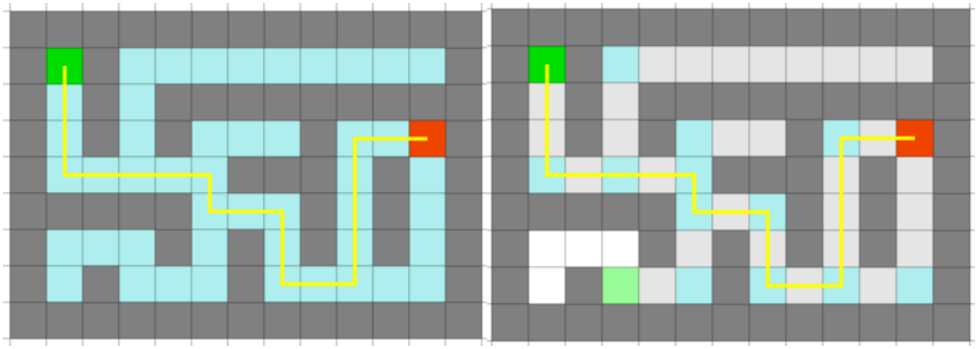
Recherches sur la manipulation

J'ai imprimé les deux documents sur la manipulation de bits fournis dans le sujet. Je me suis familiarisée avec les décalages, les tests, les masques et les mises à 0 ou 1 d'un bit.

1u << n :	décalage de n bits vers la gauche du 1
a & 0x01 :	le bit de poids faible de a est-il à 1 ?
0xEF :	masque pour 1u << 4
~(1u << 4) :	idem
a &= 0xFE :	mise à 0 du bit de poids faible
a |= 0x01 :	mise à 1 du bit de poids faible

Recherches sur la génération de labyrinthe

J'ai consulté la page wikipedia http://en.wikipedia.org/wiki/Maze_generation_algorithm et j'ai mené des tests sur http://qiao.github.io/PathFinding.js/visual/



Je pensais implémenter l'algorithme de Prim pour la génération et l'OJPS (orthogonal Jump Point Search) pour la recherche du chemin.

Algorithmes choisis

L'algorithme de Prim donnant des culs de sac courts, j'ai préféré en changer. J'ai finalement implémenté une version simplifiée de la génération par DFS (parcours en profondeur). Voici mon pseudo-code :

```
Mettre des murs partout
Pour toutes les cases du labyrinthe, faire :
    Choisir une direction au hasard
    Tant que toutes les directions n'ont pas été testées, faire :
        Regarder la case adjacente dans la direction choisie
        Si elle existe, qu'il y a un mur et qu'elle n'est pas visitée, faire :
            Casser le mur pour les deux cases
            Dire que la case actuelle est visitée
            Aller dans la case suivante
        Si non, faire :
            Tourner la direction de 90°
    Dire que la case actuelle est visitée
```

Pour la recherche du chemin, j'ai utilisé la technique populaire de « la main sur le mur » : on pose sa main gauche sur le mur à sa gauche et on avance en ne lâchant jamais le mur à sa gauche. La technique est fastidieuse (on revient souvent sur ses pas) mais permet de trouver la sortie dans un labyrinthe sans îlot. Voici mon pseudo-code :

```
Partir de l'entrée
Choisir une direction au hasard
Tant que toutes les directions n'ont pas été testées, et qu'on n'est pas sortis, faire :
    Regarder la case adjacente dans la direction choisie
    Si elle existe, faire :
        S'il y a un mur, faire :
            Tourner de direction vers la gauche
        Si non, faire :
            Dire que la case actuelle est dans le chemin
            Aller dans la case suivante
            Tourner de direction vers la gauche
```

Pour la recherche du meilleur chemin, j'ai choisi d'ajouter le même algorithme avec la main droite : on tourne à droite plutôt qu'à gauche. Ensuite, je calcule la longueur du chemin en enlevant les cases « visitées pour rien » (quand on revient sur ses pas) pour chacune des deux méthodes. Enfin, je compare ces deux chemins et je choisis le plus court. Cette méthode me fournit toujours le chemin le plus court dans un labyrinthe sans îlot.

J'ai pris soin de venir voir l'enseignant pour vérifier l'absence d'îlot dans ses labyrinthes de test.

Ce choix stratégique m'a permis de ne pas alourdir le code avec une structure supplémentaire et la gestion des listes chainées. Par ailleurs, je tire ainsi le meilleur parti des bits qui me permettent de stocker les informations nécessaires : je colle au mieux à l'esprit du projet proposé.

II. Ecriture des specs et de la structure fonctionnelle

Aspect d'une case

J'ai choisi un affichage dans le terminal pour pouvoir me consacrer aux algorithmes plutôt qu'à l'affichage.

Voici une case vide, une case du chemin final, et une case où on est revenu sur ses pas :

Choix typographique

J'ai choisi l'indentation Allman, et la case suivante :

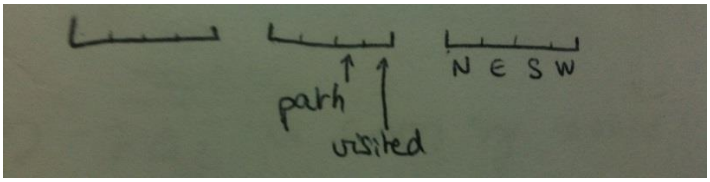
```
nom_de_fonction()
nom_de_variable ;
```

Structure maze

J'ai respecté le sujet et ai représenté le labyrinthe par une structure contenant les éléments suivants :

- Tableau 2D d'%hu
- Nombre de lignes et de colonnes
- Position [x,y] de l'entrée et la sortie

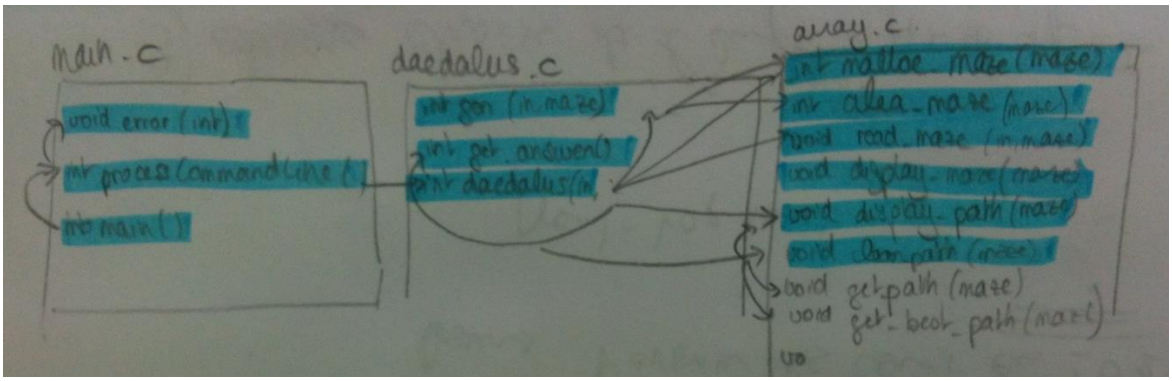
Voici mon utilisation des bits :



Je n'ai pas eu besoin de la position [x,y] du chercheur de chemin.

Fonctions

Comme on me l'a appris, j'ai beaucoup travaillé avec une feuille et un crayon sur ce projet. Voici mes specs :



Je surligne les fonctions au fur et à mesure de leur implémentation.

Implémentation de la structure fonctionnelle

Avant de me lancer dans le code fonctionnel, j'ai implémenté la structure fonctionnelle :

- struct.h : la structure qui représente le labyrinthe
- error : display le message d'erreur selon un numéro d'erreur
- process_command_line : récupère les arguments, lance le fonctionnel daedalus
- main : appelle juste process_command_line

III. Implémentation

Lecture depuis un fichier

J'ai récupéré les méta-données (taille, entrée, sortie) dans ma fonction **daedalus**, puis j'ai créé une fonction **read_maze**. Celle-ci contient une double boucle 'for' pour récupérer la valeur des cases du labyrinthe.

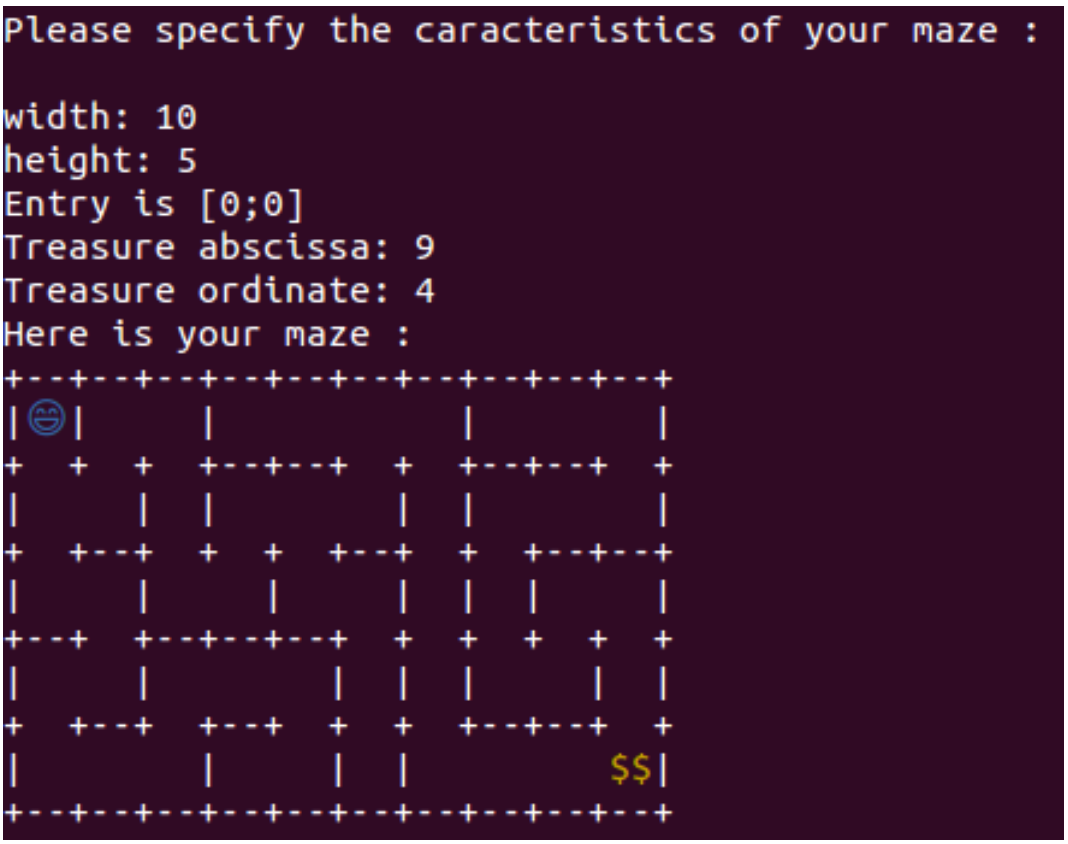
Génération contrôlée

L'utilisateur est invité à rentrer les méta-données à la main dans le terminal via la fonction **gen**. La fonction **alea_maze** prend le relais et crée un labyrinthe cohérent :

- les bords du labyrinthe ont des murs
- si un mur est présent sur une case, la case adjacente a bien ce même mur
- utilisation de mon algorithme (DFS simplifié)

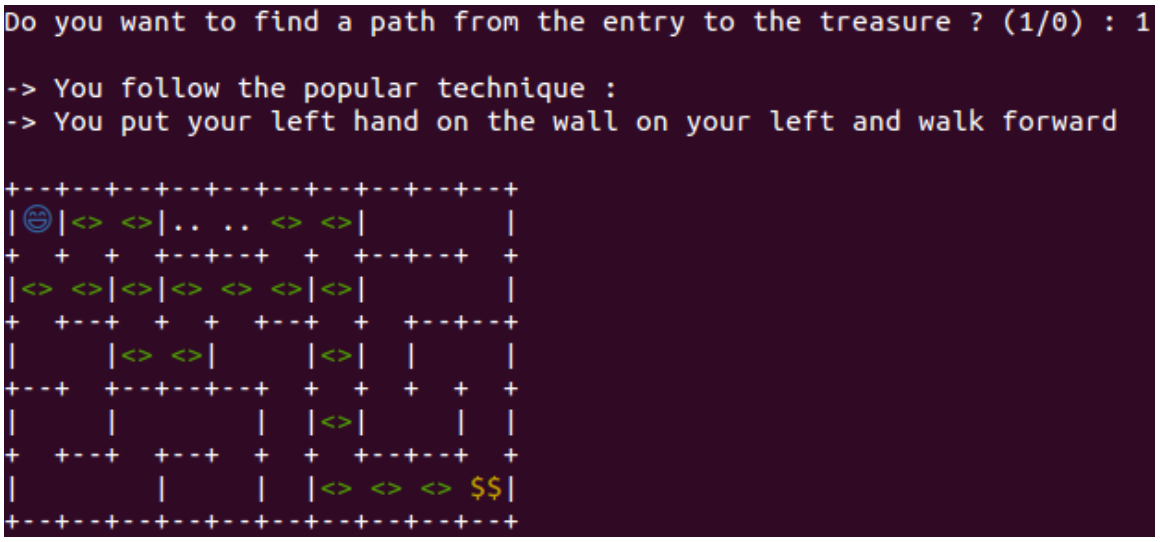
Génération aléatoire

Les méta-données sont générées par la fonction **gen**. La fonction **alea_maze** prend le relais et crée un labyrinthe cohérent.



Recherche d'un chemin

La fonction **get_path** permet de rechercher un chemin selon la technique de la « main sur le mur ». Elle prend un paramètre afin de rechercher avec la « main droite » ou la « main gauche ». Elle met à jour les bits visited et path des cases.



Recherche du plus court chemin

Comme expliqué précedemment, la fonction **get_best_path** appelle **get_path** deux fois et compare la longueur des chemins grâce à **get_path_length**.

Elle sélectionne le chemin le plus court, l'affiche dans le labyrinthe et affiche la longueur.

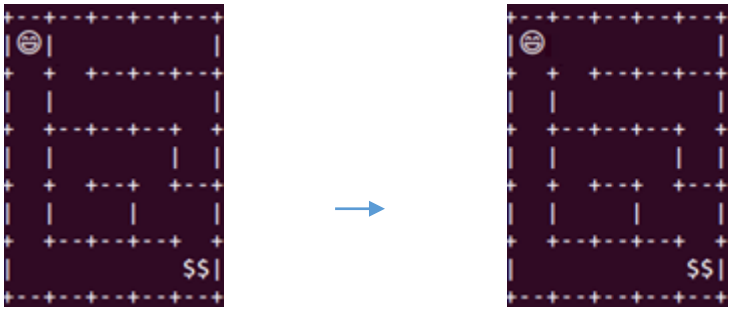
IV. Résolution de problèmes

Artefacts dans la génération aléatoire

J'ai remarqué que ma génération aléatoire pouvait aboutir au cas suivant :

J'ai donc corrigé mon algorithme en ajoutant la mise à 1 du bit de visite.

Par la suite, ma génération aléatoire pouvait aboutir au cas suivant :

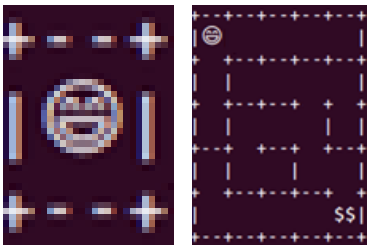


Pour minimiser la possibilité d'avoir des zones inaccessibles, j'ai supprimé les murs South et East de la case [0 ;0] quand la case [1 ;1] avait des murs North ou West

Boucle infinie en l'absence de chemin

Pour éviter les boucles infinies lors de la recherche de chemin j'ai :

- compté le nombre de changements de direction successifs (cas d'un départe emmuré)
- compté le nombre de passage sur la case départ (cas d'une sortie inaccessible)



V. Effets stylistiques

Départ

J'ai choisi d'afficher un smiley au départ, pour que l'utilisateur ait un léger sentiment d'identification :

Arrivée

J'ai choisi de transformer la sortie en trésor pour rendre l'expérience utilisateur plus ludique :

Case visitée n'appartenant pas au chemin final

J'ai choisi de faire apparaître les cases visitées lors du trajet « main sur le mur » mais non retenues avec « .. ».

Case appartenant au chemin

J'ai choisi de représenter les cases du chemin par « <> » par souci de lisibilité, après avoir testé notamment « ## », « [] » et « ** ».

Couleur

Enfin, afin d'agrémenter l'expérience utilisateur, j'ai fait le choix de l'emploi de la couleur.